On the (in)security of ElGamal in OpenPGP

Luca De Feo, Bertram Poettering and Alessandro Sorniotti
IBM Research Zürich

June 6, 2022, Security Standardisation Research Conference, Genova

Cryptographic standards, what's the worse that could happen?

- Theoretical break.
- Side-channel leakage.
- Implementations secure in isolation, do not interoperate.
- **Implementations secure in isolation, insecure when interoperating.**

OpenPGP

- An IETF Encryption standard, since 1998.
- One of two standards for end-to-end email encryption (along with S/MIME).
- Many implementations:
  GnuPG, Botan (rnp/Thunderbird), Go (Protonmail), Libcrypto++, ...
- IETF RFCs:

RFC 4880	**OpenPGP Message Format**
RFC 3156	MIME Security with OpenPGP
RFC 5581	The Camellia Cipher in OpenPGP
RFC 6637	Elliptic Curve Cryptography in OpenPGP

OpenPGP algorithms

Hash Functions: MD5, RIPE-MD, SHA-1, SHA-2.

Symmetric Ciphers: IDEA, TripleDES, CAST5, Blowfish, AES, Twofish, Camellia.

Public Key Encryption: RSA, ElGamal, ECDH.

Signature Algorithms: RSA, DSA, ECDSA.

RFC 4880 (dated November 2007)

*"Implementations MUST implement DSA for signatures, and **ElGamal** for encryption. Implementations SHOULD implement RSA [...]"*

ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

m	message
y	random

ElGamal Encryption

p	prime	m	message
$\alpha \bmod p$	generator	y	random
$\alpha^x = X$	public key		

$$(Y = \alpha^y, \quad X^y \cdot m)$$ encryption

$$m = X^y \cdot m / Y^x$$ decryption

Public key algorithms specifications in OpenPGP

RSA	PKCS #1
ECDH	NIST SP 800-56A + RFC 6637
DSA	FIPS 186-2
ECDSA	FIPS 186-3
ElGamal	**El Gamal ’85 / Handbook of Applied Cryptography ’97**

ElGamal according to the OpenPGP standard?

8.4.1 Basic ElGamal encryption

8.17 Algorithm Key generation for ElGamal public-key encryption

SUMMARY: each entity creates a public key and a corresponding private key.
Each entity A should do the following:

1. Generate a large random prime p and a generator α of the multiplicative group $\mathbb{Z}_p^*$ of the integers modulo p (using Algorithm 4.84).
2. Select a random integer a, $1 \leq a \leq p-2$, and compute $\alpha^a \bmod p$ (using Algorithm 2.143).
3. A's public key is (p, α, α^a); A's private key is a.

8.18 Algorithm ElGamal public-key encryption

SUMMARY: B encrypts a message m for A, which A decrypts.

1. *Encryption.* B should do the following:
   (a) Obtain A's authentic public key (p, α, α^a).
   (b) Represent the message as an integer m in the range $\{0, 1, \ldots, p-1\}$.
   (c) Select a random integer k, $1 \leq k \leq p-2$.
   (d) Compute $\gamma = \alpha^k \bmod p$ and $\delta = m \cdot (\alpha^a)^k \bmod p$.
   (e) Send the ciphertext $c = (\gamma, \delta)$ to A.
2. *Decryption.* To recover plaintext m from c, A should do the following:
   (a) Use the private key a to compute $\gamma^{p-1-a} \bmod p$ (note: $\gamma^{p-1-a} = \gamma^{-a} = \alpha^{-ak}$).
   (b) Recover m by computing $(\gamma^{-a}) \cdot \delta \bmod p$.

II. The Public Key System

First, the Diffie–Hellman key distribution scheme is reviewed. Suppose that A and B want to share a secret K_{AB}, where A has a secret x_A and B has a secret x_B. Let p be a large prime and α be a primitive element mod p, both known. A computes $y_A \equiv \alpha^{x_A} \bmod p$, and sends y_A. Similarly, B computes $y_B \equiv \alpha^{x_B} \bmod p$ and sends y_B. Then the secret K_{AB} is computed as

$$\begin{aligned} K_{AB} &\equiv \alpha^{x_A x_B} \bmod p \\ &\equiv y_A^{x_B} \bmod p \\ &\equiv y_B^{x_A} \bmod p. \end{aligned}$$

Ma
This
Gran
Cryp
Th
Univ
1501

In any of the cryptographic systems based on discrete logarithms, p must be chosen such that $p-1$ has at least one large prime factor. If $p-1$ has only small prime factors, then computing discrete logarithms is easy (see [8]).

Now suppose that A wants to send B a message m, where $0 \leq m \leq p-1$. First A chooses a number k uniformly between 0 and $p-1$. Note that k will serve as the secret x_A in the key distribution scheme. Then A computes the "key"

$$K \equiv y_B^k \bmod p, \tag{1}$$

where $y_B \equiv \alpha^{x_B} \bmod p$ is either in a public file or is sent by B. The encrypted message (or ciphertext) is then the pair (c_1, c_2), where

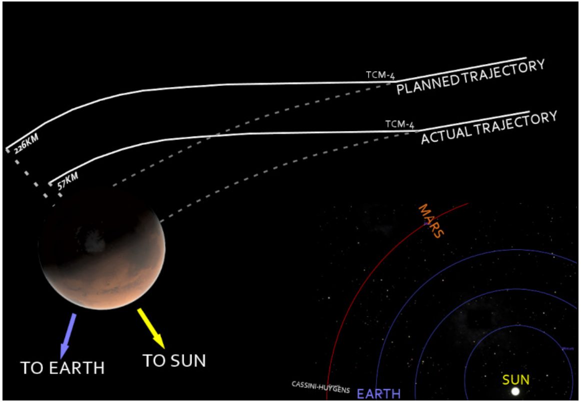
$$c_1 \equiv \alpha^k \bmod p \qquad c_2 \equiv Km \bmod p \tag{2}$$

and K is computed in (1).

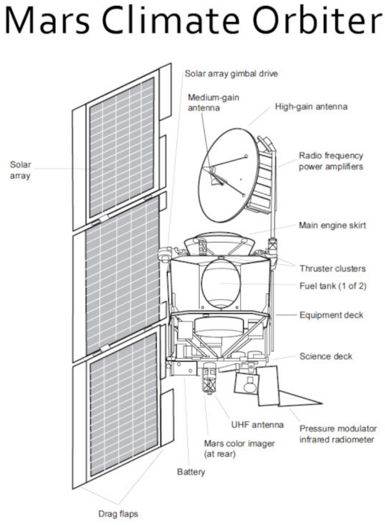
ElGamal in the wild (OpenPGP ecosystem)

Large prime p	Safe prime	"Schnorr" prime	"Lim-Lee" prime	other
Generator α	primitive element		generates subgroup	
Private key	$0 < a < p$		"short exponent" optimisation	
Ephemeral key	$0 < k < p$		"short exponent" optimisation	

What could possibly go wrong?



credit: Wikipedia



Mars Climate Orbiter spacecraft

Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is secure taken in isolation.
  - They are interoperable: functionally and securely.

Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is secure taken in isolation.
  - They are interoperable: functionally and securely.

- Go does not implement ElGamal key generation and is the least offender.

Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is secure taken in isolation.
  - They are interoperable: functionally and securely.
- We analyse 800K registered PGP ElGamal public keys:
  - 2K of them are exposed to practical plaintext recovery when GnuPG, Botan, Libcrypto++ (or any other library using the “short exponent” optimisation) encrypts to them.
    We call these cross-configuration attacks.
- Go does not implement ElGamal key generation and is the least offender.

Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is secure taken in isolation.
  - They are interoperable: functionally and securely.
- We analyse 800K registered PGP ElGamal public keys:
  - 2K of them are exposed to practical plaintext recovery when GnuPG, Botan, Libcrypto++ (or any other library using the “short exponent” optimisation) encrypts to them. We call these cross-configuration attacks.
- Go does not implement ElGamal key generation and is the least offender.
- We find side channels leaking ElGamal secret keys in GnuPG, Go and Libcrypto++:
  - GnuPG claimed to be side-channel resistant.
  - Our attack against GnuPG becomes more powerful in the cross-configuration scenario.

Prime generation

Goal: prime p with at least one large prime factor $q | (p-1)$.

Safe primes: $p = 2q + 1$:

- Considered kind of expensive, back in the '90s.

"Lim-Lee" primes: $p = 2q_1 q_2 \cdots q_r + 1$, all q_i large:

- Cheaper than safe primes,
- Protecting against the same attacks.

"Schnorr" primes: $p = 2qf + 1$, with f arbitrary:

- Cheapest,
- Popularized by Schnorr signatures, DSA, FIPS-186-2.

Random primes: risky, don't do it!

Other: your imagination is the only limitation!

Prime and exponent sizes

	GnuPG[1]			Libcrypto++[2]	DSA	RFC 3766	BSI TR 02102
$\lvert p\rvert$	"Wiener's table"	$\lvert x\rvert$	$\lvert y\rvert$	$\lvert x\rvert$, $\lvert y\rvert$	$\lvert q\rvert$	$\lvert q\rvert$	$\lvert q\rvert$
512	119	181	184	120			
768	145	220	224	144			
1024	165	250	248	164	160	135	140
1536	198	298	304	198		168	
2048	225	340	344	226	224	190	200
3072	269	406	408	268	256	224	256
4096	305	460	464	304		280	
7680	1160	1741	1744	398	384	380	384
15360	2120	3181	3184	530	512	480	512

[1] *"Michael Wiener's table on subgroup sizes to match field sizes. (floating around somewhere, probably based on the paper from Eurocrypt 96, page 332)."*

[1] *"I don't see a reason to have a x of about the same size as the p. It should be sufficient to have one about the size of q or the later used k plus a large safety margin. Decryption will be much faster with such an x."*

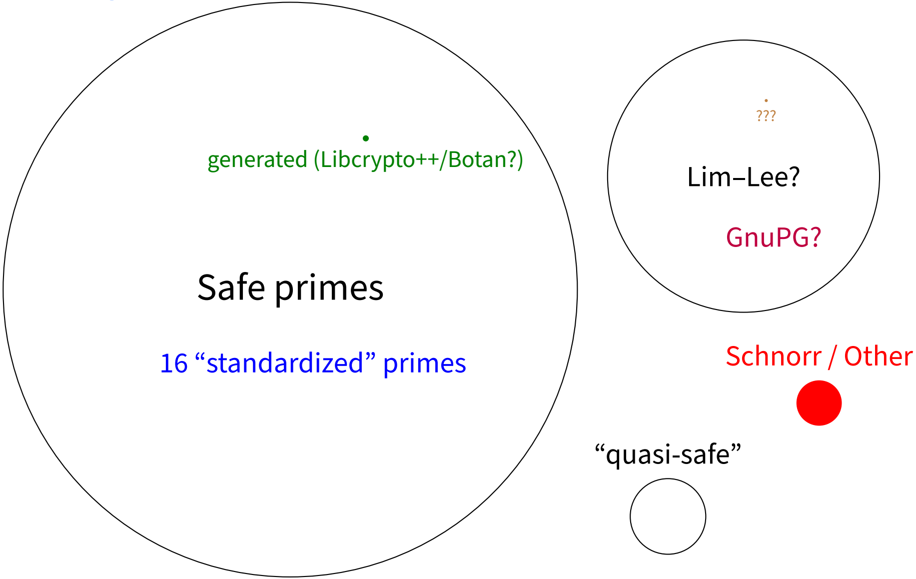
[2] *"extrapolated from the table in Odlyzko's "The Future of Integer Factorization" updated to reflect the factoring of RSA-130"*

800K registered OpenPGP ElGamal public keys

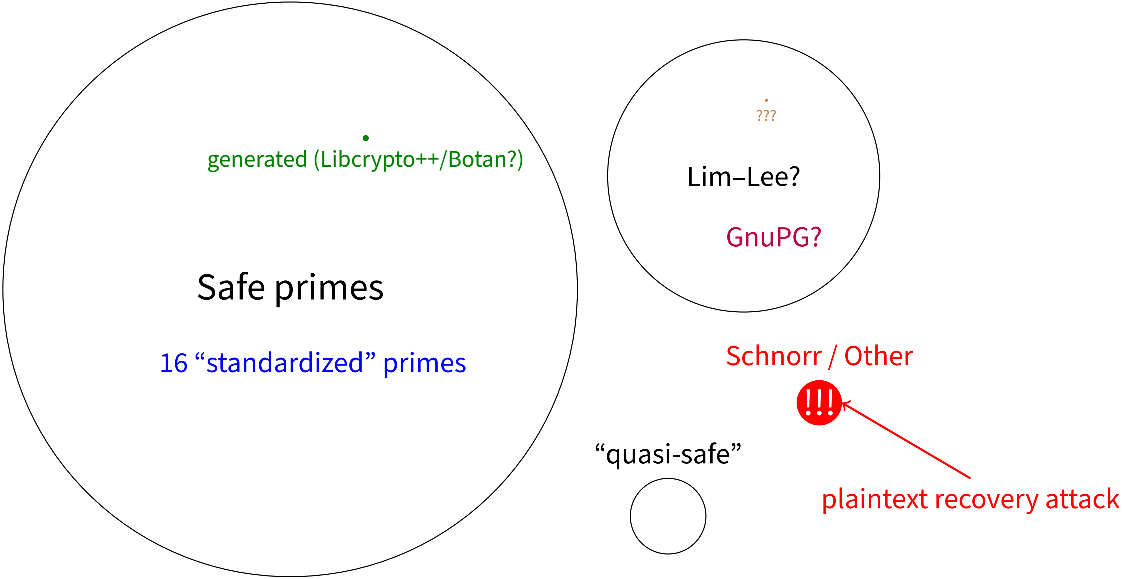
- We downloaded an OpenPGP server dump produced on Jan 15, 2021;
- Out of 2,721,869 keys, 835,144 contain ElGamal subkeys;
- For each El Gamal subkey we factored $(p-1)$ as much as we could:[3]
  - We collected the keys into prime types: safe prime, *probable* Lim-Lee, “quasi-safe”, Schnorr/other;
  - Based on factorization, we inspected the order of the generator $\rightarrow$ tentative attribution to originating software.

[3] Trial division + ECM

800K registered OpenPGP ElGamal public keys



800K registered OpenPGP ElGamal public keys



Discrete log: when α is primitive

$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$

$$\alpha^x$$

Discrete log: when α is primitive

$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$

$$\alpha^x$$

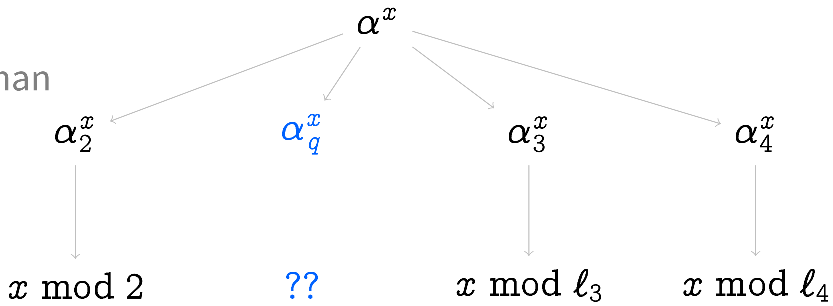
Pohlig–Hellman

$$\alpha_2^x \qquad \alpha_q^x \qquad \alpha_3^x \qquad \alpha_4^x$$

Discrete log: when α is primitive

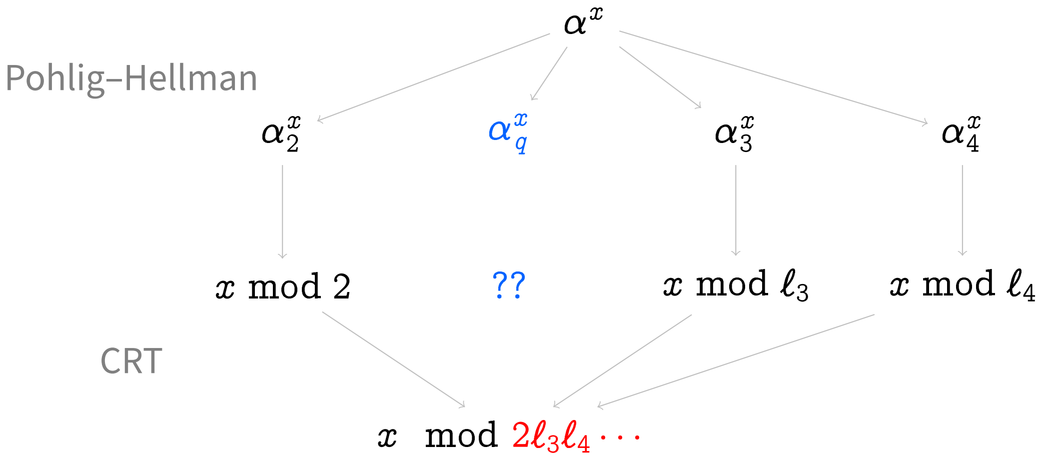
$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$

Pohlig–Hellman



Discrete log: when α is primitive and x is "short"

$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$



ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

m	message
y	random

ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

m	message
y	random

$$\left(Y = \alpha^y, \quad X^y \cdot m \right)$$ encryption

$$m = X^y \cdot m / Y^x$$ decryption

ElGamal Encryption

p	prime	m	message
$\alpha \bmod p$	generator	y	random
$\alpha^x = X$	public key		

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

ElGamal Encryption

Bingo 0: key recovery from public key only (van Oorschot–Wiener)

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

ElGamal Encryption

Bingo 0: key recovery from public key only (van Oorschot–Wiener)

Bingo 1: message recovery from single ciphertext (this work)

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are “large” primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		“short”	
$y \in$	$[1, p-1]$		“short”	

ElGamal Encryption in OpenPGP

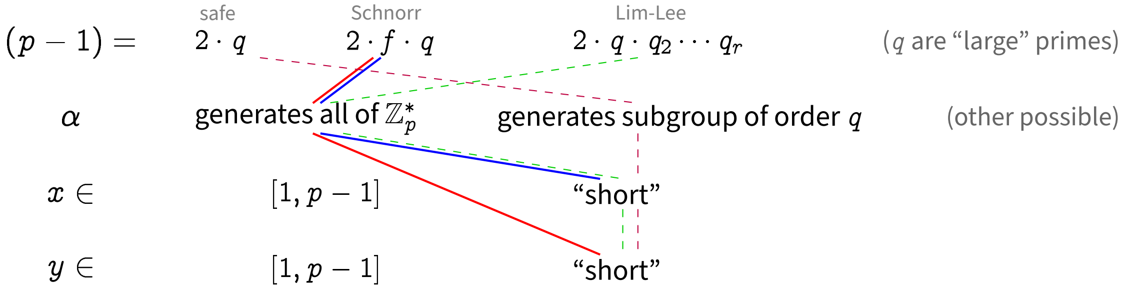
GnuPG: Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

ElGamal Encryption in OpenPGP

GnuPG: Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

Libcrypto++/Botan: safe primes, generates subgroup, short exponents.



ElGamal Encryption in OpenPGP

GnuPG: Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

Libcrypto++/Botan: safe primes, generates subgroup, short exponents.

Go : no key generation, $y \in [1, p-1]$.

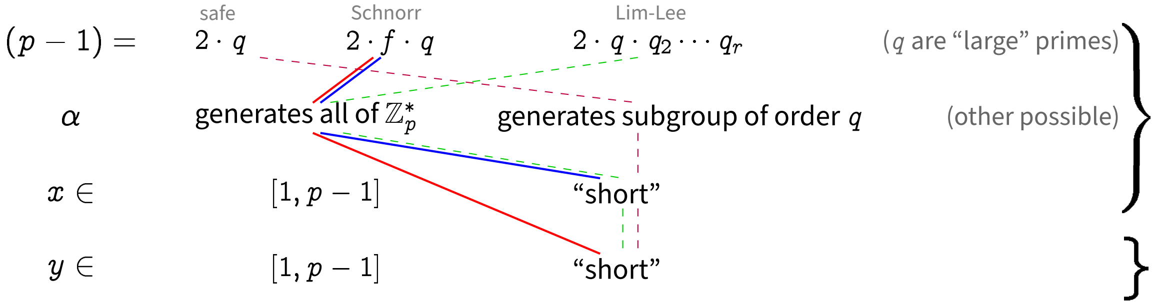
	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

ElGamal Encryption in OpenPGP

GnuPG: Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

Libcrypto++/Botan: safe primes, generates subgroup, short exponents.

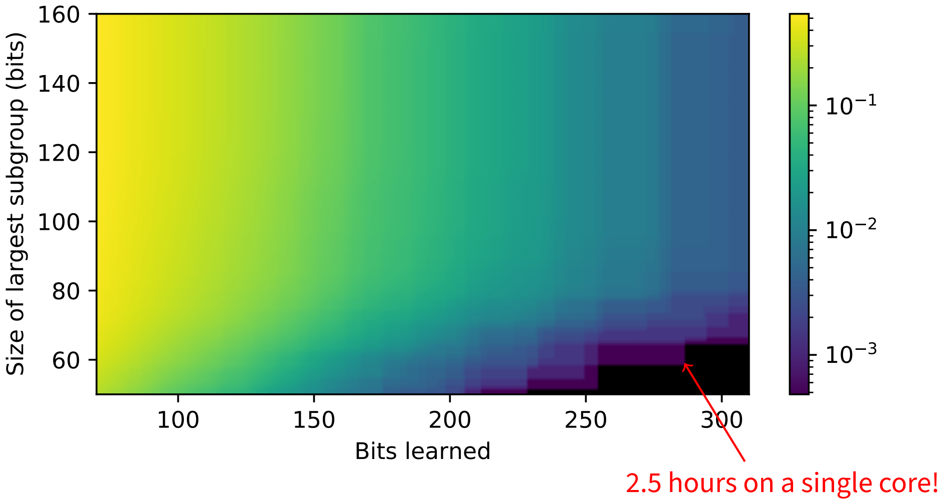
Go: no key generation, $y \in [1, p-1]$.



800K registered OpenPGP ElGamal public keys

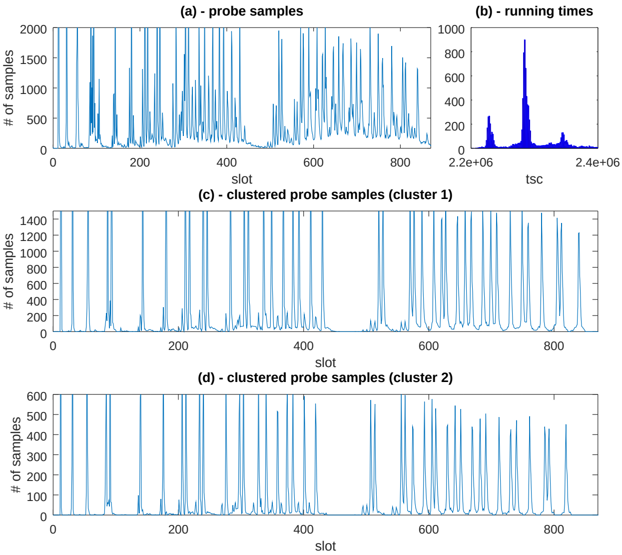
prime type	group size			quantity	
	$p-1$	q	other	total	since 2016
Safe prime I	x			472,518	783
Safe prime II		x		107,339	219
Lim–Lee I	?			211,271	6,003
Lim–Lee II		?		47	24
Quasi-safe I	x			15,592	89
Quasi-safe II		x		20	3
Quasi-safe III			x	26,199	125
Schnorr I	?			828	810
Schnorr II		?		27	26
Schnorr III			x	1,304	1,300

How bad is it?



160
140
120
100
80
60
Size of largest subgroup (bits)
100
150
200
250
300
Bits learned
10^{-1}
10^{-2}
10^{-3}
2.5 hours on a single core!

Side channel vulnerabilities in exponentiation (GnuPG)



Side channel vulnerabilities in exponentiation $\rightarrow$ Key recovery

Threat model

- Co-located attacker;
- Targets the exponentiation in the decryption routine;
- Must trigger decryption (e.g., email decryption).

Techniques FLUSH+RELOAD (instruction cache), PRIME+PROBE (data cache).

Findings

Key \ Library	Libcrypto++	Go	GnuPG
$x \in [1, p-1]$	trivial	easy	unfeasible
GnuPG	trivial	easy	unfeasible/state
Libcrypto++/Botan	trivial	easy	state/commodity*

*Verified experimentally on 2048 bits key.

How did we get here?

1991 Phil Zimmerman creates PGP[4]

PGP Marks 10th Anniversary

5 June 2001 - *For a signed version of this announcement click here*

Today marks the 10th anniversary of the release of PGP 1.0.

It was on this day in 1991 that I sent the first release of PGP to a coupl

1996 PGP 3 adds support for DSA and ElGamal, mainly to avoid patents.

1998 RFC 2440, first OpenPGP standard.

> *“The generator and prime must be chosen so that solving the discrete log problem is intractable. The group g should generate the multiplicative group mod p-1 or a large subgroup of it, and the order of g should have at least one large prime factor. A good choice is to use a* ***"strong" Sophie-Germain prime*** *in choosing p, so that both p and (p-1)/2 are primes. In fact, this choice is so good that* ***implementors SHOULD do it****, as it avoids a small subgroup attack.”*

[4] Rerieved from `https://www.philzimmermann.com/EN/news/PGP_10thAnniversary.html`

How did we get here?

1997 Lim & Lee publish *“A key recovery attack on discrete log-based schemes using a prime order subgroup”* at CRYPTO.

“The purpose of this paper is to point out the insecurity of various discrete log-based schemes using a prime order subgroup. More specifically, we present a key recovery attack on these protocols, which can find all or part of the secret key bits. Our attack is closely related to the choice of parameters and the checking of protocol variables. Thus, as is usual, our attack, once identified, can be easily prevented by adding suitable checking steps or by using ’secure’ parameters.”

1997-1999 First release of GnuPG.

“I bought some of the LNCS volumes when I started with gpg in 97 and the Lim-Lee algorithm looked like an very efficient way to create large and safe primes.”

Werner Koch, private communication

How did we get here?

1995 Crypto++ implements ElGamal

2002 Botan implements ElGamal

> *“For compatibility with GnuPG, ElGamal now supports DSA-style groups”*
>
> *Version 1.1.1 release notes*

2007 RFC 4880, current OpenPGP standard. All technical bits on ElGamal dropped.

Question to you:

Could this happen again?

More info, get in touch, …

Luca: @luca_defeo

Ale: @sigusr0

Blog: `https://ibm.github.io/system-security-research-updates/`

Paper: `https://ia.cr/2021/923`

Thank you!